Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

MLOps Scaling ML in an Industrial Setting

Author: Yizhen Zhao (2658811)

1st supervisor:	Dr. Adam S.Z. Belloum	University of Amsterdam
daily supervisor:	Gonçalo Maia da Costa	Dashmote B.V.
2nd reader:	Dr. Zhiming Zhao	University of Amsterdam

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

July 13, 2021

"I am the master of my fate, I am the captain of my soul"
from Invictus*, by William Ernest Henley*

Abstract

Machine learning has evolved from an area of academic research to a real-word applied field. This change comes with challenges, just like any other fields, gaps and differences exist between how it works in an academic setting and what is required in a real-word configuration. Following continuous integration, development and delivery practices in software engineering, similar trends have happened in rapid development of machine learning (ML) features, called MLOps. The goal of this paper is to build a MLOps framework and practices that facilitate and scale the machine learning lifecycle in an industrial setting, also the reproducibility and traceability of machine learning projects are guaranteed. This MLOps framework also acts as a template for people or teams who are interested in building their own MLOps framework. It inherits practices from DevOps and introduces other practices that are unique to the machine learning system, for example, data versioning. Our MLOps practices automate the entire machine learning lifecycle, bridge the gap between development and operation, enable better collaboration and communication between different teams who are operating machine learning systems.

Keyworks: *MLOps, DevOps, Machine Learning, Industrial Setting, Data Versioning, Continuous Development*

Acknowledgements

This six-month Master project was carried out as an internship at Dashmote, in the Data Product team. This internship was a great opportunity for me to apply both my professional and academic skills. I would like to start by thanking Dashmote for giving me this opportunity to work with them on this interesting project. I would like to express my sincere gratitude to my daily supervisor, Gonçalo Maia da Costa for his trust, guidance, feedback and all the support during the project. Also my amazing team mates, Samuel Kellerhals, Evgenia van Rijn, etc., for their inspiration and feedback. I would also like to thank Dr.Adam S.Z. Belloum for all the opportunities he gave me and guidance on completion of the Master project. Furthermore, I would like to thank Dr. Zhiming Zhao for giving the valuable insights for this thesis.

Finally, many thanks to my family and close friends for their encouragement and support throughout my study.

Contents

List of Figures

Glossary

AES ECS Amazon Elastic Compute Cloud, page 37

API Application Programming Interface, page 8

AWS Amazon Web Services, page 8

Azure ML Azure Machine Learning, a cloud-based service for creating and managing machine learning solutions, page 18

Azure ML SDK It can be used for Python to build and run machine learning workflows with Azure ML service, page 25

CI/CD The combined practices of continuous integration, either continuous development or continuous deployment, page 39

CLI Command-Line Interface, page 32

CPU Central processing unit, page 24

DevOps A set of practices that combines software development and IT operations, page 2

DNS Domain Name System, page 33

EC2 Elastic Compute Cloud, a web service that provides secure, resizable compute capacity in cloud, one of the services offered by AWS, page 34

ECR Elastic Container Registry, one of the services offered by AWS, page 10

ECS Elastic Container Service, a container orchestration service that helps you deploy, manage and scale containerized applications, offered by AWS, page 34

ID Identification, page 33

Jenkins A free and open source automation server. It helps automate the parts of software development related to building, testing, and deploying, facilitating continuous integration and continuous delivery, page 16

KPI Key performance indicator, page 16

ML Machine Learning, page 1

NL The Netherlands, page 17

PODs The smallest, most basic deployable objects in Kubernetes. A Pod represents a single instance of a running process in your cluster, page 11

pull request An notification to team members that a developer completed a feature and request team members to review the work, page 21

RDS Relational Database Service, one of the services offered by AWS, page 16

REST REpresentational State Transfer, is an architectural style for providing standards between computer systems on the web, making it easier for systems to communicate with each other, page 11

S3 Simple Storage Service, one of the services offered by AWS, page 8

SQL A programming language, designed for managing data held in a relational database management system, page 7

SQLAlchemy A library that facilitates the communication between Python programs and databases, page 29

UI User interface, page 9

unit testing A type of software testing where individual units or components of a software are tested, page 16

URI Uniform Resource Identifier, page 30

URL Uniform Resource Locator, page 10

1

Introduction

We live in a data-driven world. The volume of data is exploding, a study[1] shows over 2.5 quintillion bytes of the data is generated every day. There is an information explosion happening in human history. It might be because of the rise of smartphones and all kinds of digital devices. This explosion of data leads to a circle of data analysis, which in turn leads to data creation, data analysis and new insights. Organizations and governments are trying to derive values from these data by adopting advanced analytics. This enables better decision-making to optimize the value of their services or business.

Artificial Intelligence (AI) and Machine Learning (ML) are projected to become the mainstream technologies in the coming years. Machine Learning offers a powerful toolkit for helping us solve complex real-world problems, making use of the massive data, turning insights into action and providing useful information, either in the academic research field or in industrial innovation. Machine learning in academic research and in the real-world machine learning system, may sometimes mean different things. Sculley et al.(1) shared his point of view in his paper that machine learning code is only a small fraction in the real-world machine learning system. The required surrounding infrastructure is complex and usually, it takes longer to deploy ML in production[2], compared to developing it. Some other tasks need to be taken into consideration, such as maintaining the problem of traditional code plus additional ML-specific issues, integrating the ML system into current industrial setting, running and maintaining several ML models in production, etc.

Following the slogan of the company, Dashmote[3], "Turn Data into Actions", we try to unlock the power of data, provide global brands with computer vision-based services.

[1] https://www.takeo.ai/can-you-guess-how-much-data-is-generated-every-day/

[2] By *production* we mean an environment where code/apps/services are deployed and available for users.

[3] https://dashmote.com/

Through advanced data analytics and making data-driven solutions to help our clients, like Heineken, Coca-Cola, to better connect with the market around them and win in the marketplace with what Dashmote offers. We also introduced machine learning to help us solve complex real-world problems and provide useful information for our clients.

MLOps, or DevOps for machine learning, is becoming a necessary skill set for enterprises to leverage the benefits of machine learning in the real world. It is a practice for better collaboration and communication between the data scientists team and data engineers team to improve the automation of the entire machine learning lifecycle and deploy it in the production environment. To some extent, MLOps will enable better collaboration and communication between teams in organizations, speed up the delivery time of projects and reduce the labor work needed for machine learning projects. For instance, some labor work like manually training and retraining the model, keeping track of the experiment results, manual releases and deploying the model, etc.

This thesis aims to take a machine learning project at Dashmote[1] and develop a prototype that applies MLOps practices to the machine learning lifecycle. It helps us to understand the problems and challenges when applying ML in an industrial environment and to better manage the entire machine learning lifecycle, which includes model and data versioning, keeping track of experiment results so that the machine learning project has reproducibility and traceability, model monitoring and model deployment in production.

The rest of this thesis is organized as follows. In Chapter 2, I present the background information about machine learning in production and outline the generalized problem statement regarding the corresponding thesis topic, followed by research questions. In Chapter 3, I will describe the state-of-art relevant to this thesis. Then, the proposed ML lifecycle design and MLOps architecture design are introduced in Chapter 4, followed by the implementation details of this graduation project prototype. In Chapter 5, the discussion, reflection and future work of this graduation project are presented. The summary of this paper is in Chapter 6. Last but not the least, terminologies mentioned in this paper are explained in Glossary and code examples are listed in Appendix.

[1] https://dashmote.com/

2

Background

2.1 Machine Learning in Production

We have conducted a literature review on the machine learning system in production(2), where we intend to find out challenges and difficulties in building and maintaining ML systems in a production environment, and why MLOps are introduced into the marketplace. ML in production focuses more on engineering. Not only focus on implementing ML models but also build the whole ML-related infrastructure within an industry. We summarize the importance and the challenges facing Machine Learning in production (MLOps) in the following section as they are very relevant to the work presented in this thesis.

2.1.1 Challenges in Machine Learning Projects

As a machine learning system is not the same as the traditional software system, a set of ML-specific challenges needs to be taken into consideration. I summarize the challenges into following levels:

- Dependency level

  Developing ML models relies on several components, such as data, ML algorithm code and/or parameters. During experimentation, these components might change overtime which leads to different versions. Different versions will lead to different model behaviors, a certain version of data, code and parameters generates a certain model. Having several versions of data, code, parameters in a ML project, but without certain strategies to handle different versions, can make us lose control of our ML system. Creating a frozen version of the data and parameters can help us

keep track of different versions. But versioning carries its own costs, for example, maintaining multiple versions over time.

Traditional software systems utilize modular design to maintain the whole system, you can change one part without interfering with others. While compared to traditional software systems, ML systems have no such clear boundaries between components. ML pipelines work with data versions, algorithm code versions and/or different parameters. Any change in these components mentioned before triggers new model versions.

- Management level

  ML systems in the production environment, usually, the goal is running and maintaining dozens or hundreds of models simultaneously, which leads to managing challenges. For example, how to monitor the whole production pipeline? How to update or assign the power configuration for the model?

  To automate and accelerate ML lifecycle management, we need to have a strategy or workflow that helps us keep tracking the data, hyperparameters, experiment results, etc. To some extent, it helps better management and collaboration. Manual logging the experiment process is not efficient and error-prone.

2.1.2 Importance of MLOps and Difference with DevOps

MLOps, which is DevOps for ML. It is a set of practices used to streamline the ML lifecycle. MLOps and DevOps do have some similarities, they share the same research method and research cycle. Some DevOps practices are also applied in MLOps. For instance, they both encourage and facilitate collaborations between data scientists, engineers and operations. They both emphasize process automation in continuous development and continuous delivery into production.

DevOps helps data engineers build, maintain and improve the entire service lifecycle. While MLOps is mainly for data science. MLOps helps data scientists manage and maintain the iterative ML lifecycle in an efficient and controllable way. ML workflow is more complex as it includes several ML-specific fields, such as collecting datasets, building models, tuning hyperparameters, etc. Not only code needs versioning, but also ML artefacts (e.g. data, models and metrics) so that the ML system has reproducibility.

2.2 Problem Statement

Machine learning is used in many of the services we are using in our daily activities, and lots of companies are using ML to find the hidden pattern behind data. However, there are lots of problems we might encounter when applying machine learning in a production environment. I summarize the most important problems into the following points (more details can be found in (3)(4)(5)).

- Machine learning is not only about the code, but also data. One of the characteristics of machine learning is that ML development is iterative and experimental and data scientists might need to do different experiments to optimize a metric (e.g. prediction accuracy), tuning parameters and generating features. This might lead to hundreds of different versions of ML algorithm code, data, hyper-parameters or experiment results. Managing all the experiment settings, code versions, results, turns out to be a big problem.

- Machine learning pipeline contains several steps, namely, collecting data, generating features, training models, and deploying models that can be used in production. Without an automated ML pipeline and centralized place where the development teams keep the ML artefacts and the experiment results, it is difficult for teams, who are planning to use ML in production, to manage the entire ML pipeline. ML artefacts might be placed in different places, experiment results are kept and tracked in personal documentation or logs. Such an approach is problematic when it comes to manage the communication and collaboration between the teams, and also inefficient for teams to deliver a product with good quality as fast as it can.

- A machine learning model can only deliver added value to an organization when it's available to users or other systems and deployed in production. While machine learning systems in the real-world are complex and present different sets of challenges, some of which are shared with traditional software services, and some are ML-specific challenges. DevOps introduces a set of practices in engineering that focuses on techniques and tools to maintain and support existing production systems. While there are still some problems unique to productizing machine learning, for instance, there is no agreed best practice to handle the whole machine learning lifecycle in production.

2.3 Research Questions

Based on the problems we encountered with ML projects in production, the research questions I defined and the rational motive for the stated research questions are:

- What are the challenges in developing machine learning projects in production?
  - I wish to identify the challenges we might meet when developing and deploying ML in production.
- Why MLOps? What are the differences between MLOps and DevOps?
  - I wish to present the importance of MLOps in ML projects and the difference between DevOps and MLOps.
- How to handle data versioning issues in machine learning projects and how to deploy ML models into production efficiently?
  - I wish to present the state-of-art relevant to data versioning and model deployment in ML projects.
- What kind of MLOps frameworks or strategies that we could learn from when integrating ML systems into the industrial setting?
  - I wish to identify the frameworks or architectures in the marketplace that apply MLOps principles.

3

Related Work

In this chapter, the state-of-art relevant to this thesis is presented and we discuss the relevant research work on various subjects. We classify and group the selected literature based on the topic. Research questions for this paper are listed in Section 2.3. The research and related work for each research question will be discussed in the following subsections.

3.1 Data Version Control

Git[1] is a software version control system that has been widely used on the marketplace. Inspired by Git, Anant introduced two tightly-integrated systems in their paper(6), an open source tool[2], one is dataset version control system (DSVC) that enables data scientists to capture their modifications, identify different versions and share datasets. Another one is DATAHUB, a platform built on top of DSVC, which allows data scientists to perform data analysis, data cleaning, and visualization. There are some relational database-based version control mechanisms, but they rely highly on the data schema. While in reality, datasets often consist of mixed data structures. DSVC is similar to git but supports richer query languages. Compared to github, datahub also provides more features for working with data. Their proposed DSVC systems aiming to provide the following features:

- It can handle large datasets (100s of MBs to 100s of GBs) and also a large number of versions.
- It supports querying specific versions of dataset (e.g.using SQL), analyzing differences between versions, and exploring information across datasets.

[1] https://git-scm.com/
[2] https://datahub.io/

- It supports data analysis and exploration. For instance, providing information about when a record was last modified.
- It provides trigger-like functionality, so-called hooks, similar to git hooks, automatically triggers customized actions. For providing features like automated analysis, keep data products up-to-date, etc.

The core of DSVC is the dataset version control processor (DSVCP), which manages and processes different versions. DSVCP exposes versioning API for client applications. The version query processor (VQP) is integrated with DSVCP and exposes VQL, an enhanced version of SQL that allows users to query datasets, as an interface for clients to use. Datahub runs on top of VQP and DSVC, as a server that clients used to store the data, also provides versioning API and VQL to applications. Versioning API provided by DSVC is similar to Git API, including the commands such as **create** (create a new dataset), **branch** (create a new version of dataset), merge (merge branches), **commit** (commit local changes to dataset), **rollback** (revoke changes) and **checkout** (create a local copy of branch). The two tightly-integrated systems, drawing an analogy with git and github, provides the possibility for users to deal with dataset versioning challenges. It can be used to manage data in machine learning projects as well.

There are some other competitive data versioning tools mentioned in this online blog(7). DVC, data version control, will be introduced and used in this graduation project, in Section 4.4. Pachyderm[1], a data science and processing platform with built-in data versioning and lineage. The idea of version control in Pachyderm works similarly as Git but with some exceptions. It deals with plain text, binary files and large datasets. A centralized repository exists and your data is continuously updated in the master branch of the repository. You can work with a specific data commit in a separate branch. Unlike Git, we do not store a copy of the repository in the local projects. Hence, the merge conflicts (like we meet in Git very often) do not occur.

AWS Sagemaker[2] not only provides mode deployment (will be introduced in this graduation project as well, in Section 4.5) but also data labeling. AWS Sagemaker Ground Truth[3] is a data labeling service that provides accurate training dataset for machine learning. It provides custom or built-in data labeling workflow that supports a variety of use cases including images, video, text, etc. The original data is stored on AWS S3, then a labeling job can be created by using custom or built-in workflow. After that, labelers can use the

[1] https://www.pachyderm.com/
[2] https://aws.amazon.com/sagemaker/
[3] https://aws.amazon.com/sagemaker/groundtruth/?nc1=h_ls

labeling UI with assistive labeling feature to label the dataset. Now an accurate training dataset is ready for use.

3.2 ML Model Deployment in Production Environment

How to deploy ML models into production in an efficient way is another challenge in ML projects. There are different ways of deploying ML models into production. AWS Sagemaker[1] is a fully managed machine learning service and it is a great place and a reliable way to start if you want to deploy the ML models into production quickly. Using AWS Sagemaker for model deployment is also our choice in this graduation project, described in Section 4.5. There are also many other competitive tools or technologies to deploy models:

Azure[2] is another cloud provider, providing its own way of deploying ML models(8). The workflow is similar no matter where you deploy your models. First is to register the model that you want deployed. Secondly, the code, an entry script, that will be used in the web service, for performing the predicting on input data. Thirdly is to define an inference configuration. It describes the Docker[3] container and all the files within your project source directory to use when deploying the web service. Then it defines a deployment configuration. It specifies the resources needed for your web service, such as the amount of memory and cores in order to run the web service. Finally the model is ready to be deployed and requests can be sent to check the status of your deployed model. Azure provides a full MLOps cycle for machine learning projects, from training to deploying. We utilize Azure ML to train the model in this graduation project, described in Section 4.6, which is the team's decision. For organizations and teams who use Azure as the main cloud provider, they can stick with Azure for managing the full ML lifecycle.

MLflow[4], an open source for ML lifecycle, also supports model deployment. MLflow is also introduced in this graduation project, but our main focus is on MLflow tracking, in Section 4.7, which tracks and keeps all the ML experiment results. Some other features of MLflow, such as MLflow Model Registry[5], and Python APIs[6] like `mlflow.azureml` and `mlflow.sagemaker` can be used together to deploy the model to custom serving tools. MLflow Model Registry allows you to register the trained model in a centralized place, with

[1] https://aws.amazon.com/sagemaker/
[2] https://azure.microsoft.com/en-us/
[3] https://docs.docker.com/get-started/overview/
[4] https://mlflow.org
[5] https://www.mlflow.org/docs/latest/model-registry.html
[6] https://www.mlflow.org/docs/latest/python_api/index.html

model version control. Based on this online blog(9), `mlflow.sagemaker`[1] works similarly to our Sagemaker batch transform, in Section 4.5. It needs the details of the AWS account to have the permission set up. The model file can be provided by either MLflow, if model is registered with MLflow, or stores the model on AWS S3 and provides the S3 path. Then provide the docker image URL, the docker image hosted by AWS ECR. You can customize the ML algorithms in the docker image. As shown in these tutorials(10)(11) from Azure[2] and Databricks[3], `mlflow.azureml`[4] helps to register the MLflow model (model registered on MLflow, local model path or AWS S3 path) with Azure ML and build a custom image on Azure ML for model deployment. This image can be deployed as a web service to Azure Container Instances (ACI) or Azure Kubernetes Service (AKS).

3.3 MLOps Framework for Building Integrated ML System in Production

Emmanuel(12) introduced an edge MLOps framework for edge Artificial intelligence Internet of Things (AIoT), which is a system that facilitates edge computing[5] for AIoT applications. This MLOps framework enables continuous delivery, development and monitoring of machine learning models at the edge for AIoT applications.

The framework describes an end to end, fully automated machine learning pipeline. They utilize Azure machine learning[6] to execute the end-to-end ML pipeline. Started from continuous fetching data from edge devices into cloud storage. Azure Blob storage[7] is used for storing all the data collected from sensors via each device and ML models. During the ML experiment, first versioning the data that will be used in training so that the experiment is reproducible, then model training, evaluation. Finally packaging and registering the model and waiting for deploying to edge devices (i.e. in production). Azure DevOps[8] is used to maintain and version control the ML algorithm code used for building ML models, and then build and release ML artefacts, models to edge devices and perform needed jobs. Every day a release is triggered to monitor the edge devices to check the model

[1] https://www.mlflow.org/docs/latest/python_api/mlflow.sagemaker.html#module-mlflow.sagemaker
[2] https://azure.microsoft.com/en-us/
[3] https://databricks.com
[4] https://www.mlflow.org/docs/latest/python_api/mlflow.azureml.html
[5] Edge computing is the process of performing computing tasks physically close to devices, rather than in cloud.
[6] https://azure.microsoft.com/en-us/services/machine-learning/
[7] https://azure.microsoft.com/en-us/services/storage/blobs/
[8] https://azure.microsoft.com/en-us/services/devops/

performance. Based on the performance to determine whether to deploy an alternate model or not. Model retaining is also based on the previous output. In the previous step, if an ML model is replaced with another one, then the replacement model is retained on the cloud with real-time data. If performance of the new model improves, then the retrained model will be registered and waiting for future deployments.

In Pölöskei's paper(13), they described the main idea behind several cloud-based MLOps pipelines. Cloud technology makes computing more affordable and manageable. MLOps apply the same approach as DevOps, but mainly focus on the model development for eliminating the personnel and technology gap in the development.

- Workflow as graph

  DAG (directed acyclic graph) can be used to describe the pipeline workflow, and a pipeline can operate on the cluster. Each job is executed as a DAG node. Each task can be grouped by components, like python functions. They can interact through the inputs, outputs and can be reused by other components.

- GPU based pipelines

  The training process in ML projects in industry can be resource consuming, but this resource should be granted on demand. Training a deep learning model might take some time, local settings might not be able to support the training process efficiently. Cloud computing is a valid option for this kind of use-case.

- Tensorflow Extended (TFX)

  Tensorflow is the state-of-the-art deep learning framework. TFX library helps TensorFlow model development in production. A standard end-to-end TensorFlow pipeline in TFX provides a high-performance application in cloud-native environments, and can be managed by Airflow or Kubeflow.

A case study is introduced in this paper(13), where the authors present a prototype of MLOps pipeline in KubeFlow cloud-native environment. The workflow triggers container level cloud-native architecture based on the repository. Data preparation is the first and most time-consuming step, it includes preprocessing, data digestion, etc. Version control is also required because during experiments, existing models could have better performance than the new release. Then the model building happens in dedicated PODs. After evaluation and model selection, the final model will be deployed to the REST endpoint. The acceptance of MLOps allows rapid research loops, therefore, the pipeline and model can be arranged in production efficiently.

4

Project Implementation

In this chapter, we first introduce the proposed machine learning lifecycle design, including the main function of each component. The idea is to give an overall understanding about the end-to-end ML pipeline. Then we display the MLOps architecture design for this graduation project. This prototype helps to automate the ML lifecycle in terms of model development and model deployment. First we introduce the overall design of the MLOps architecture, the main function of each part is included and how they work together as a whole. Then the proposed ML development process is presented in order to provide a standard workflow for our ML projects to follow. The implementation details of necessary components within this architecture are described in the following sections. The reflection on the outcome of each component will be discussed and the metrics we defined to highlight the contribution of each component and compare the situation before and after the introducing of this component in our MLOps framework.

4.1 Machine Learning Lifecycle Design

Figure 4.1 presents an overview of the proposed ML development lifecycle. To illustrate the ML development lifecycle we will use the machine learning use cases developed at Dashmote, which is called flag-combo. It is a classification task, which tells whether a meal is a combo meal[1] or not. If it is a combo meal (e.g. burger with fries), then set the value of flag combo as 1 for that meal. In the following sections we will present the main function of each step of the ML lifecycle.

[1] A combination meal, often referred as a combo-meal, is a type of meal that typically includes food items and a beverage

The emphasis on the current project is on the last four steps, model development and deployment. The prototype of the ML lifecycle design and implementation details is explained in the next section, Section 4, aiming to address the problems we proposed in Section 2.2.

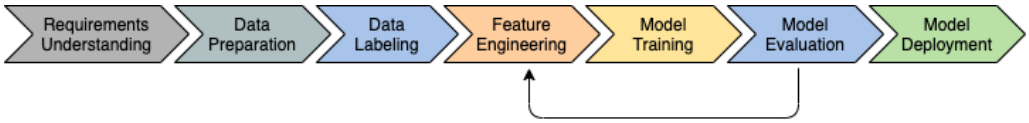

Figure 4.1: Proposed ML lifecycle

4.1.1 Understanding the Requirements

The first step is to understand the business requirement, what are the behaviours we expected from this model and what features are needed. It is important to understand the requirements of the business side before actually implementing the ML model. In this case, we wish this model can classify whether a meal is a combo meal or not. Features like price, name and meal description for that meal are needed since they can tell what packages are inside that meal.

4.1.2 Data Preparation and Data Labeling

The most time-consuming parts are the data preparation and data labeling. Data scientists are responsible for creating the dataset for training and validating the ML model, as well as validating the dataset if there are any requirements on data quality. To some extent, the data quality decides the quality of the model.

Meal data is collected from various online food delivery platforms, such as Ubereats[1], Deliveroo[2] etc. Each record in the dataset represents a single meal, with price, name, description for that meal, etc. During *Data Preparation*, the labeling set is created from the meal data. Then in *Data Labeling*, we manually label the dataset based on the definition of combo meal we agreed upon in the last step, *Requirements Understanding*. Creating a manual label column which specifies whether it is a combo meal or not. This column will be included as a feature to train the model. In the real-world scenario, the number of non-combo meals is way more than the combo meals. In order to have a non-biased training set, we might need to create more data and label them or create some fake combo

[1] https://www.ubereats.com/nl-en
[2] https://deliveroo.nl/nl/

meal data based on the real cases. For instance, a fake combo data can be a combination of the name and meal description from other two real combo records. The methodology of making fake data needs to be varied, otherwise the model can be biased and cannot see the whole picture of the real meal situation.

4.1.3 Feature Engineering, Model Training and Model Evaluation

In *Feature Engineering*, data scientists or ML developers start working on ML algorithms, selecting features, tuning parameters, etc. Then during *Model Training* and *Model Evaluation*, the model is trained and model performance is evaluated. The whole development process is iterative and experimental, several experiments are executed until the satisfying results are obtained. During the process, data and ML artefacts (e.g. trained models) are versioned so that we can guarantee the reproducibility and traceability of ML projects. ML experiment results are managed in a good manner. The methodology of versioning data is described in Section 4.4.

The implementation of this classification model is not the main focus of this graduation project. The idea is to use price, name and meal description as features to train the model since we believe, to some extent, these features can tell us whether the meal is a combo meal or not. We expect the model to classify the unseen data based on those features. The improvement of the model performance, further feature engineering will continue in the near future, which will be conducted by other data scientists in the team. Therefore, in this thesis, we will not explain the details about model development.

Apart from developing the ML models, data engineers are responsible for integrating the ML systems into current industrial settings. Build an architecture that manages the whole ML lifecycle with the goal of automating the whole process, guaranteeing the reliability of ML projects, with at least manual efforts within the process as possible. The prototype of dealing with the ML development process is presented in Section 4.

4.1.4 Model Deployment

ML model that is not deployed into the industrial environment to generate value is only a costly experiment. From the last step, we already received the trained model with satisfying performance, versioned and released a new version for it. Then the model is ready to be deployed in the production environment and to perform prediction logic on unseen data. The description of model deployment into production relevant to this thesis

project is presented in Section 4.5. With the prototype we proposed, presented in Section 4, no extra handover is required from model development to model deployment.

4.2 MLOps Architecture Design

The overall design of our MLOps architecture is shown in Figure 4.2. This architecture also describes our MLOps workflow, how we handle the model development process and model deployment within our industrial settings. With the help of Git[1], DVC[2] and MLflow[3], we guarantee the traceability and reproducibility of ML projects. Two cloud providers are introduced to help us scale the model training process and apply prediction logic in the production environment efficiently.

Before having this MLOps framework, there was no mechanism to manage the whole ML lifecycle. No mechanism to version control the dataset that has been used in training the model, which makes our life harder when we are trying to reproduce a model in a certain state. The ML experiment process and results are different according to the person who operates the ML experiments. ML models are also not version controlled and released in a good manner. No standard way of deploying trained models into production at all. After this MLOps framework is built, the dataset and all ML artefacts (e.g.model) are carefully tracked and version controlled. The ML experiment process is standardized and results are kept in a centralized place. No manual handover is required for model deployment as this MLOps framework bridges the gap between development and operation.

This architecture contains two main parts: ML model development (*Step 1, 2, 3, 4*) and model deployment (*Step 5, 6*). Model development happens on the left side, starting from the git repository, *dash-ml-flag-combo*, which contains all necessary ML algorithm code for developing the model. Within this git repository, DVC is used to help us keep tracking of the data we used in ML project (*Step 1*). Features and implementation details of data versioning with DVC is described in Section 4.4. After that, we build ML models, generate features, and tune parameters (*Step 2*). We provide the options to execute ML jobs (e.g.model training) in a local environment or leverage cloud resources (i.e.Azure Machine Learning), in case the ML job needs extra compute power. Then, the experiment results (e.g.metrics, parameters) and any kinds of ML artefacts (e.g.models) for each ML job are logged and tracked by MLflow tracking, a centralized place where all the experiment results are kept for each experimental run (*Step 3*). MLflow tracking provides a user-friendly

[1] https://git-scm.com/
[2] https://dvc.org/
[3] https://mlflow.org/

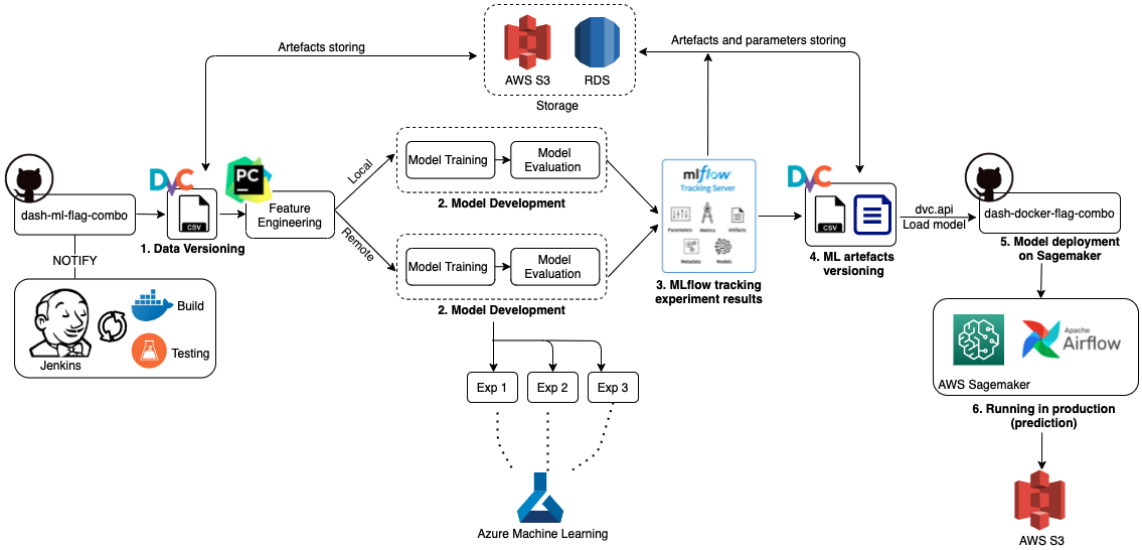

Figure 4.2: MLOps architecture

UI that allows you easily view the experiment details of your ML development process. The iterative experiment process ends until we receive the satisfying result (e.g.model performance reaches the teams' KPI). Then we use DVC to version control the final models and any other ML artefacts. Finally, using Git version control to release a new version/tag[1] for that model in that specific state (*Step 4*). All the dataset, models and ML artefacts are stored on AWS S3, experiment results (e.g.metrics, parameters) are stored on RDS[2], as long as they have been tracked by DVC and/or logged into MLflow tracking. During the development, Jenkins[3] is used for code unit testing, automatic building whenever a git commit is made, pull request is created or branches are merged and continuous deployment.

After obtaining the trained model and necessary artefacts that will be used in production, we plan to deploy them, the process happens on the right side of Figure 4.2. Another git repository is used, called *dash-docker-flag-combo*, which helps us to prepare the docker image with prediction logic and inference code that will be used in AWS Sagemaker to start the batch transform job. Then apply the prediction logic on unseen data and finally store the transformed data into AWS S3 *(Step 5, 6)*. The model and other artefacts (if any) used in production are loaded during runtime via DVC API[4] because of *Step 4*. Since

[1]Git tag allows you to identify specific release versions of your code. Official document about Git tag: https://git-scm.com/book/en/v2/Git-Basics-Tagging

[2]https://aws.amazon.com/cn/rds/

[3]https://www.jenkins.io/

[4]https://dvc.org/doc/api-reference

models and other artefacts are tracked by DVC using DVC file[1], and DVC files are version controlled by Git, therefore, DVC API allows you to access the models that under DVC and Git control in repository *dash-ml-flag-combo*. Implementation details of model deployment with AWS Sagemaker and DVC API usage for loading models files are in Section 4.5. Airflow[2] is used here to help us automatically triggering the model deployment pipeline.

4.3 Refine ML Development Process

In this section, we describe the details of our ML development workflow. Within our current strategy, different versions of datasets and models are safely tracked by DVC. All the experiment runs/results are collected in a centralized place, provided by MLflow. They are easy to manage and trace. ML development can be done in different environments. It is normal for developers to develop models in the local environment first and then scale out to the cloud. Git is responsible for code version control and code, model release. These three tools work together to guarantee the traceability and reproducibility of ML projects. Jenkins helps in continuous integration and continuous development, it runs autonomous tasks we defined, such as code unit testing, data testing, docker image build, etc.

First our Git repository structure is given, with brief introduction of essential components. Then our proposed workflow of dealing with the ML development process is described.

4.3.1 Git Repository Structure

The proposed git repository structure is listed below, the essential components are:

- assets: It includes the country-based (i.e. NL) datasets and all other ML artefacts (e.g. model). We classify our assets by country.
- dvc_pipeline: Country-based DVC pipelines, DVC pipeline setup has been introduced in Section 4.4.3. Having separate DVC pipelines allow us to work on one of them, without interfering others.
- scripts/`run_pipeline.py`: The entry point to trigger the ML pipeline process.
- `executor.py`: An executor to execute different pipelines, either execute the ML jobs in a local environment (i.e. local pipeline (`local_run.py`)) or utilize cloud resources (i.e. remote Azure pipeline (`azure_run.py`)).

[1] https://dvc.org/doc/user-guide/project-structure/dvc-files#dvc-files
[2] https://airflow.apache.org/

- pipeline: A place where different pipelines are set up. Current options are local pipeline and Azure remote pipeline. It provides the flexibility for users to execute ML jobs in different environments. If another cloud provider is introduced (e.g. AWS), a python script for configuring the AWS pipeline can be added separately.
- `azure_executor.py`: The executor for triggering and submitting ML jobs to cloud environment (i.e. Azure ML).
- ml_pipeline: It contains the python scripts for building and evaluating the ML model.
- Dockerfile: The customized docker image setup for creating the environment for running ML jobs on Azure ML. The details are introduced in Section 4.6.2.

```
dash-ml-flag-combo/
├── assets
│   ├── data
│   │   └── NL
│   │       ├── NL_training.csv
│   │       └── NL_validation.csv
│   ├── model
│   └── metrcis
├── dvc_pipeline
│   └── NL
│       ├── dvc.yaml
│       ├── dvc.lock
│       └── params.yaml
└── src
    └── dash_ml_flag_combo
        ├── ml_pipeline
        │   ├── train.py
        │   └── validate.py
        ├── pipeline
        │   ├── local_run.py
        │   └── azure_run.py
        ├── scripts
        │   └── run_pipeline.py
        └── utils
            └── azure_executor.py
```

```
    executor.py
Dockerfile
```

4.3.2 ML Development Workflow

The relationship and workflow between those components mentioned in Section 4.3.1 is visualized into Figure 6.2. We use python scripts here to represent each step. The python script can contain a python class, or a set of python functions.

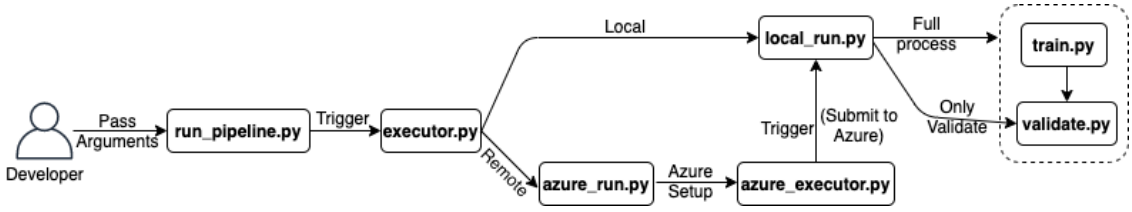

Figure 4.3: ML development internal workflow

Figure 4.2 and Figure 6.2, display the overall architecture and workflow within the ML development process. The end-to-end developing workflow is described in the following steps:

- If you are working with a new Git repository without any DVC setup, the first step is to install DVC and initialize it in your Git repository. Section 4.4.1 provides the details about DVC initialize. If you are working with an existing Git repository with DVC setup, you can follow the Gitflow[1] to create your own feature branch and work on this branch for further implementation. Use `dvc pull` to synchronize the datasets and other artefacts, download them from remote storage.
- Use `dvc add` to add dataset into DVC control if you are adding new datasets or updating datasets. As mentioned in Section 4.4.2, Section 4.2 *Step 1*, DVC uses so-called DVC files to keep track of the data. For example, the DVC command: `dvc add assets/data/NL/NL_training.csv` will generate the corresponding DVC file for the NL training data. The same rules apply for other datasets.
- Create country-based DVC pipelines (in Section 4.4.3) that runs each stage of the ML project. We create two base stages in DVC pipelines that execute in the local environment, stage `train_validate_local` (i.e.full process, train and validate) and stage `validate_local`. It is true that in ML development, whenever a model is

[1] https://guides.github.com/introduction/flow/

trained, the validation step is always followed so that the model performance is known. There might also be a case that we only want to re-validate the model (e.g. a new validation set is added), then we create a separate stage only for validation purposes. By having country-based DVC pipelines instead of one DVC pipeline that contains all the stages of different countries, it provides a clear classification between different sub-projects. We can work on one sub-project (e.g. NL developing) without interfering with others. The example code of creating stages in DVC pipeline in this case is presented in Appendix A.1, DVC Build Pipeline Examples. The generated DVC files (i.e. *dvc.yaml*, *dvc.lock*) are presented in Appendix A.2, DVC Pipeline Examples as well.

- Triggering the ML pipeline process by either directly executing Python script or using DVC command `dvc repro` to kick off the stages in DVC pipelines. The process starts by user, passing necessary arguments to `run_pipeline.py`. Then it triggers the `executor.py` for executing ML pipeline process either in local environment or utilizing cloud resources (i.e. on Azure) based on the arguments user passed in.
  - If it is in a local environment, the local pipeline (i.e.`local_run.py`) is run. Then it triggers either a full process (i.e. train first, then validate) or only validate process.
  - If it requests a cloud environment, the azure pipeline (i.e.`azure_run.py`) is run. It creates all necessary configuration that will be needed to execute ML jobs on Azure, then the `azure_executor.py` will kick off the local pipeline with Azure setup on cloud environment, execute either full process job or only validate job. It is always the local pipeline that is run (i.e.`local_run.py`), since both local and Azure environments are using the same training and validating python scripts, Azure just requires different settings. Configure ML jobs to Azure is described in Section 4.6.
- During each experiment run, MLflow tracking, which will be described in Section 4.7 is responsible for logging all the necessary results, such as metrics, parameters used, even models and any other artefacts. Using MLflow tracking with ML experiments will be discussed in Section 4.8.
- The iterative experiment process ends when the developers get the model with satisfying results. Then we use DVC command `dvc add` to version control the final version of ML artefacts (e.g. models). Now the model is version controlled by DVC, as well as the data we used that the model has been trained on.

- Before letting Git to version control everything, using `dvc commit`, `dvc push` to commit DVC pipelines, DVC files to DVC and push dataset, artefacts to remote storage. Then let the Git version control all the changes and results by using `git add`, `git commit`.
- At this point, a pull request can be opened, notifying the team members that a feature or a ML development process has been completed. After team members' reviewing, this *feature* branch is going to be merged into the *develop* branch and create a new release. Then a ML development process has been completed.

4.4 Data Versioning with DVC

As mentioned in section 2.1 and 2.2, data versioning is a big challenge in a machine learning project. Machine learning experimentation is an iterative process, therefore, different versions of data, code and other ML artefacts (e.g.models) are generated through the process. How to manage the different versions? How to make sure the traceability and reproducibility of ML projects?

DVC[1] is a tool that makes ML models shareable and reproducible. It works similarly as Git and designs to handle large files, models and any other ML artefacts. In the course, Industrial Internship[2], I took one of the machine learning projects at Dashmote, implemented a prototype of data and ML artefacts versioning in the ML project. The details of the implementation can be found in the report(14) and the online blog(15) I published. The reasons why we choose DVC are, firstly, it works similarly as Git, providing easy-to-use commands. We work with Git on a daily basis for traditional software system development. Secondly, it can easily integrate with Git-based projects, work together with Git to help us version control the dataset and the code. Last but not the least, it is an open source tool. The outcome of this course about DVC is summarized in the following subsections, it builds the foundation for this graduation project as it solves one of the biggest challenges, data versioning, in machine learning projects.

Before there is no mechanism for us to version control the dataset that has been used in ML projects. Dataset we used for training some legacy models no longer exists, which makes us lose the reproducibility of certain models. Now with this data versioning mechanism, the dataset has been taken care of DVC and kept on remote storage. The summary of implementing data versioning with DVC is presented in the following sections. In this

[1] https://dvc.org/

[2] https://studiegids.test.vu.nl/en/2020-2021/courses/XM_405080

graduation project, we extend the work to integrate the data versioning and workflow mechanism we implemented before with other services.

4.4.1 DVC Initialize

In order to use DVC and DVC features properly within your ML projects, the first step is to install and initialize it[1]. Then assign a remote storage[2] (e.g. Google drive[3], AWS S3) to DVC, where the datasets will be stored. Therefore, all the data files or any other files can be kept in this remote storage instead of keeping them in the Git repository.

4.4.2 DVC Features

DVC allows you to version control your data files and other ML artefacts. It provides a so-called ***.dvc** file[4] to version control the data files, which contains a unique md5 hash that uniquely identifies your data files. Then Git is responsible for version control of the code and that DVC file. DVC has the following features when versioning the data files:

- **Versioning data**: It provides a simple command, `dvc add`, which allows you to add data files into DVC control. It will generate the ***.dvc** file for each data file. Then Git helps you version control that ***.dvc** file.
- **Storing versioned data to remote storage**: DVC supports several remote storages, such as Google Drive[5], AWS S3[6] and etc. Versioned data files can be stored remotely, instead of keeping them on Git[7]. By setting remote storage for your ML project, you can push your data files remotely to the cloud.
- **Retrieving data files**: Having a DVC-controlled data file stored remotely on the cloud, it can be downloaded to a local project when needed. Since the DVC file, which contains a unique hash that uniquely identifies the data file, is located in the local project, DVC knows where to find the data file and download it to the local project.
- **Making changes on data**: When making data changes locally, DVC allows you to always track the latest version of your data. The command is the same as in the first

[1] https://dvc.org/doc/command-reference/init
[2] https://dvc.org/doc/command-reference/remote#remote
[3] https://www.google.com/drive/
[4] https://dvc.org/doc/user-guide/project-structure/dvc-files#dvc-files
[5] https://www.google.com/drive/
[6] https://aws.amazon.com/s3/
[7] https://git-scm.com/

step, **Versioning data**, by doing so, DVC will update the unique hash in the DVC file, then a new hash is generated for the new version of data.

- **Switching between versions**: You might have different versions of data that are tracked by DVC and Git. DVC allows you to switch between versions. Each hash in the DVC file can identify a certain version of data, and Git versions each DVC file, hence, by checkout a certain state of Git and then DVC checkout to synchronize data, you can have a certain version of DVC file in your local project.

4.4.3 DVC Helps Building ML Pipeline

DVC also supports building a simple ML pipeline for your ML project. DVC pipelines allow you better organize projects and reproduce the workflow and results later.

DVC uses *stage* to represent each single data process. It provides a *dvc.yaml* file[1], in which one or several stages are presented, for example, training stage, validation stage etc. Inside each stage, it specifies its dependencies (e.g. input data file), outputs (e.g. expected output model file) and commands that are used to run the script. Once the stages are presented in the dvc.yaml file, and located in your ML projects, stages can easily be reproduced by simple DVC command. According to the stages information in dvc.yaml file, it knows where to find the input files and what are the output files.

Another file, *dvc.lock*[2], is generated together with dvc.yaml file, which helps to record the state of the ML pipeline(s) and track its outputs. It is similar to the DVC file mentioned in Section 4.4.2, which uses md5 hash to link each file that presents in the stage. The hash for a single file will be updated if that file changed in the project (e.g. update to a new version).

4.5 Model Deployment with AWS Sagemaker Batch Transform

ML model starts providing value to the enterprise when it has been deployed into the industrial environment, which means deployed into production. But how to deploy models in production at scale and how to make prediction more efficient? There are lots of tools or services that are aiming to deploy ML models into production in an efficient and scalable way. Amazon Sagemaker[3] is a fully managed service that supports ML model building,

[1] https://dvc.org/doc/user-guide/project-structure/pipelines-files
[2] https://dvc.org/doc/user-guide/project-structure/pipelines-files#dvclock-file
[3] https://aws.amazon.com/cn/sagemaker/

training and deploying. One of the features provided by Sagemaker, batch transform[1], is a high-performance and high-throughput method for transforming data and generating inferences. It is ideal for dealing with large datasets.

The implementation details can be found in the same report(14) and the online blog(16) I published. We leverage AWS Sagemaker batch transform to deploy our trained models into production. The idea of Sagemaker batch transform is using a simple API to run prediction on the large or small batch dataset. There is no need to split the dataset into multiple chunks or run prediction in real-time, which can be expensive. Sagemaker provides a set of parameters that allows you to customize your prediction function. For instance, by customizing the payload size of your batch transform job, it will load as much as records within that payload size in the dataset and perform prediction on that mini-batch.

Sagemaker batch transform performs efficiently even with a large dataset. Within our testing, we utilize an instance with 8 CPU, 32 GiB memory to perform prediction on a 1.8G JSON file. It only takes 18 minutes to finish the prediction and this type of instance costs only $0.461 per hour[2].

By standardizing the model deployment workflow, we can have a standard way of deploying model into production environment and it can be used by all our ML projects, which guarantees consistency. Without this strategy, developers might deploy ML models into production manually, in their local environment, or using cloud resources but in his/her own way. It is difficult for teams to monitor the model that has been deployed into production as there is no standard way of doing it and no KPIs are provided to measure the process. In this graduation project, we integrate this model deployment process into the new MLOps architecture. With the help of DVC and Airflow, model deployment can be executed automatically and without any handover process.

4.6 Productize ML Experiment Process Remotely on Cloud

It is a common thing that ML development first starts in a local environment, and then scales out to a cloud environment, to utilize cloud resources. One of the advantages of cloud-based services is that it gives developers or organizations access to high-performance infrastructure that they can pay for what they use. ML applications require a ton of compute power that can be very expensive. It is more affordable to rent access to those cloud-based services than to purchase them outright.

[1] https://docs.aws.amazon.com/sagemaker/latest/dg/batch-transform.html
[2] https://aws.amazon.com/sagemaker/pricing/

In this section, we introduce Azure Machine Learning[1] to help us scale out our ML development process. As mentioned in Section 4.2 and 4.3, our infrastructure allows users to submit ML training jobs to the cloud and leverage cloud resources. The configuration details of submitting ML jobs to Azure ML will be elaborate in the following subsections. By having this set up in our ML projects, we provide the option for developers to utilize the cloud resource. In case the developers have no capability of doing ML developing in their local environment, they can still use cloud resources to fulfill the requirements of ML jobs.

4.6.1 Azure Machine Learning

Azure Machine Learning (Azure ML) is a cloud-based service for building and managing ML solutions. It helps data scientists or ML developers with end-to-end ML lifecycle and allows you to work with any kinds of machine learning, from classical ML to deep learning, supervised and unsupervised learning. Azure ML also supports different programming languages, Python or R, and with options to work with Azure ML SDK[2] or Azure ML studio[3].

Azure ML also provides all possible tools that will be needed by data scientists and ML developers for their machine learning workflows. For instance, it provides Jupyter notebooks as data scientists might be more familiar working with Jupyter notebooks. But also it supports deploying your own Python scripts to Azure ML for model building. Many other services are integrated with Azure ML as well, such as data storage, virtual machines, MLflow, Kubeflow, etc.

4.6.2 Deploy ML Jobs to AzureML

Azure ML allows you to deploy your own ML jobs to a cloud-based environment. Before actually submitting your ML jobs to Azure, there are a set of configurations that need to be created. Basically there are six steps to follow and the prerequisites are having a Azure subscription and Azure ML SDK installed. These six steps work together to provide an environment that ML jobs (i.e.model training) can run on with customized requirements. The main function of each step is described below and the corresponding Python code example for creating each step is presented in Appendix B.1, Azure ML Configuration Code Examples.

[1] https://azure.microsoft.com/en-us/services/machine-learning/
[2] https://docs.microsoft.com/en-us/python/api/overview/azure/ml/?view=azure-ml-py
[3] https://docs.microsoft.com/en-us/azure/machine-learning/overview-what-is-machine-learning-studio

- **Azure ML Workspace**

  Azure ML Workspace is a logic container that manages all ML assets, such as compute instances, data storage, pipelines, models, etc. It is the place where your ML jobs happen and the foundation of running ML jobs. The workspace can be created either through Azure Portal[1] or via Python code, Azure ML SDK.

- **Compute target**

  Compute target is an environment that allows you to train the ML models. It provides a variety of resources and developers can choose based on their needs. A compute target can be your local machine or a cloud resource. It is to pay for what you use. You can request the resource when you need it and stop when you are not using it anymore. The price varies based on the configuration of the compute target. There are four different choices of compute target[2] and we are using the compute instance which can be easily created. By choosing the virtual machine type, whether CPU or GPU, how many cores, how many Gigabytes RAM, and how many Gigabytes storage, you can customize your own compute instance. Price for that type of virtual machine per hour is given.

- **Experiment**

  Each ML run or ML job is called an experiment on Azure ML. All the information related to that run will be logged into one experiment. Experiment runs are identified by an unique experiment run ID. The purpose of this step is to assign a name to your ML jobs. By having an experiment name for your ML projects, the ML runs can be grouped together by the experiment name.

- **Environment**

  Environment is an encapsulation of the environment that includes all the necessary Python libraries and packages that will be used in ML jobs. It is the place where machine learning jobs happen. Azure ML provides curated environments that include common libraries and packages that will be needed in machine learning, but also it supports you building your own docker image and using that docker image as your environment.

  We use the Dockerfile (mentioned in Section 4.3.1) to install all the Python libraries and packages needed for machine learning. Then build the docker image locally and

[1] https://azure.microsoft.com/en-us/features/azure-portal/

[2] https://docs.microsoft.com/en-us/azure/machine-learning/how-to-create-attach-compute-studio

push the docker image to remote container registry (i.e. Azure Container Registry (ACR)). Then during the ML runs, this docker image will be loaded from ACR as the environment for machine learning jobs.

- **Script Run Configuration**

  We have Azure ML workspace ready, compute instance created, experiment and environment are available, then the next step is to connect them all together and tell Azure ML what are the configuration for the ML job, what is the *project folder* that includes all necessary Python scripts, Python classes and where is the *Python script* that execute the ML job (e.g model training).

- **Submit the Experiment**

  With all the steps above finished, the final step is to actually submit the ML job to Azure. When the ML job is submitted to Azure ML, an experiment with the experiment name we created will display in the Azure ML workspace, inside that experiment, all the necessary information related to that ML run are clearly logged. The *Python script* works as the entry point to trigger the ML job process, a snapshot of the *project folder* is created and sent to the compute target, where the machine learning happens.

  When the ML job finishes, there are two special folders *./outputs* and **./logs** are created by Azure ML. Any output files (e.g.model) exported from the ML jobs can be written to the *./outputs* folder and the ML artefacts (i.e.model) exported to this folder can also be downloaded to local projects. *./logs* folder keeps the process logs that allow developers to check.

4.7 Productize MLflow Tracking Server

MLflow is an open source platform for managing machine learning lifecycle. It allows tracking ML experiments, guaranteeing the reproducibility of ML projects. Also including model deployment and model registry. It has four main components:

- **MLflow Tracking**: It is an API and a user-friendly UI that can record and query ML experiments. It provides a centralized place where keeps all the ML experiments results. MLflow Tracking is one of the main focuses of this graduation project.

- **MLflow Projects**: It is a format for packaging data and ML codes into a reusable and reproducible way. Then it includes an API or command-line tools for executing

the ML projects. DVC pipelines, mentioned in Section 4.4.3, have the similar function as MLflow Projects. The command `dvc repro`, mentioned in our development workflow Section 4.3.2, allows us to execute ML pipelines. Hence, we stick with the DVC for executing ML pipelines.

- **MLflow Models**: It is a standard format for packaging ML models that can be used in various downstream tools. It provides a file that describes all the flavors of the model. For instance, how the model has been saved? which library is used? which version of the library specifically? (e.g. sklearn with the version 0.19.1). One of the advantages of using MLflow Model is that the standard flavors describe how to run or load the model as a Python function in any downstream tools.

- **Model Registry**: It is a centralized model store and has the similar function as MLflow Tracking. Model Registry keeps all the models that have been registered by developers. It provides model lineage, which means each version of model is linked to the experiment or run that produced this model. Also model versioning and stage transitions. For example, transferring the model stage from staging to production. As long as the model has been registered, it can be loaded as a Python function in any downstream tools by using MLflow API, *load_model()*[1], by providing the model name and the version.

In this graduation project, we will only use **MLflow Tracking**. One of the biggest challenges for us is lacking a centralized place and a strategy to manage and keep all the experiment results. Without MLflow Tracking, our ML projects require manual logging or use Git to version control experiment results (e.g.model accuracy) as well, which is not very user friendly and error-prone. **Model Registry** is a nice-to-have feature to us and compared to our ML development workflow, it will lead to a slightly different workflow methodology. This part will be discussed in Section 5. The UI of MLflow Tracking and Model Registry is displayed in Appendix C.1.

In this section, the four different MLflow Tracking working scenarios are briefly introduced, together with our choice of implementing MLflow Tracking. We first try to implement it in local environment, after that we migrate to cloud. A short introduction of Azure ML built-in MLflow tracking service is also presented and followed by the reasons why we do not choose this option.

[1] https://mlflow.org/docs/latest/python_api/mlflow.pyfunc.html#mlflow.pyfunc.load_model

4.7.1 MLflow Tracking Working Scenarios

The first thing is to understand how the MLflow runs or experiment runs are logged and data are stored. MLflow runs can be recorded to local files, to a SQLAlchemy compatible database or a remote tracking server(17). There are two components used for storage: **backend store** and **artifact store**. Backend store is mainly for storing MLflow entities (experiment runs, parameters, metrics, etc), while artifact store is for artifacts (models, files, images, etc). There are four common scenarios:

- MLflow on localhost (S1): MLflow tracking can be run on a local machine. In this case, both backend store and artifacts store are local file systems, *./mlruns*, all the ML entities, artifacts related to each run will be stored into this directory. The advantage of this scenario is that it is easy to use and deploy, there is no extra setup for integrating MLflow tracking into local ML projects.
- MLflow on localhost with SQLAlchemy compatible database (S2): As mentioned before, MLflow also supports using databases for storing ML entities. In this scenario, it runs on a local machine but with a SQLAlchemy compatible database, such as SQLite, Postgresql, etc.
- MLflow on localhost with tracking server (S3): Similar to the first scenario but a tracking server is launched on localhost. In this case, the local file directory is the default file store. However, the command `mlflow server <args>` can be used to configure what backend and artifact store are used.
- MLflow with remote tracking server (S4): MLflow also supports distributed architectures, where the tracking server, backend and artifact store can be remote hosts. In this case, a remote tracking server can be launched. The advantages of remote tracking server are that team members can access this tracking server if they have permission and the experiment runs exist on MLflow Tracking are shareable. We choose this scenario as a remote MLflow tracking server allows better collaboration and communication between different teams.

4.7.2 Azure ML Built-in MLflow Tracking

Azure ML provides built-in MLflow Tracking service(18), by installing Python package *azureml-mlflow*, it provides the connectivity for MLflow to access Azure ML workspace. MLflow tracking uses Azure ML workspace as the backend and artifacts store. All the

ML entities and artifacts that have been logged into MLflow tracking when running ML jobs with Azure ML, they will be stored in corresponding *Experiment* run, one of the components in deploy ML jobs to Azure ML, Section 4.6.2.

In this scenario, we do not need to follow the four common scenarios mentioned in Section 4.7.1. MLflow tracking works as part of Azure ML, as long as Azure ML is the cloud provider for executing ML jobs. The advantages of using Azure ML built-in MLflow tracking are that it is easy to deploy and requires little effort. But on the opposite site, this option bundles with Azure ML. If Azure is not used anymore in the future or teams migrate to another cloud provider (e.g. AWS), the MLflow tracking service will not be available anymore. In our case, our main cloud provider is AWS, therefore, we choose S4, *MLflow with remote tracking server*, Section 4.7.1, as we wish to have a separate MLflow tracking server and ML experiments are always available on it no matter which cloud provider we use.

4.7.3 MLflow Tracking Server Local Implementation

We first tried to implement an MLflow tracking server in a local environment using docker, and tested it with local ML experiment jobs. The purpose is to understand how MLflow tracking server works and to make sure it can work as a separate service. Then test with remote ML experiment jobs (i.e. on Azure ML), to make sure it can also integrate with Azure ML.

As described in S3, Section 4.7.1, a tracking server is launched on localhost and the command `mlflow server <args>` is used for configuring backend and artifact store. Since we use AWS S3 for storing datasets for DVC and it does not require extra setup in local environment, therefore, for the local implementation, we decided to use AWS S3 as our artifacts store, a PostgreSQL database running in docker as the backend store and localhost as the MLflow tracking URI[1]. The local implementation and testing procedure has two parts:

- Local setup with local script: The purpose of this step is to set up the MLflow tracking server in the local environment and use a simple Python script to test if metrics, parameters and artifacts can be logged into MLflow tracking.
- Local setup with remote ML jobs (i.e.on Azure ML): It is to test if the Azure ML can work with local MLflow tracking server. There might be some connectivity issues

[1]It is the URI for a remote server, set by MLflow function *mlflow.set_tracking_uri()*.

(e.g. connectivity between two cloud providers) and we plan to tackle those problems in the local environment first.

4.7.3.1 Local Setup for MLflow Server

The artifact store can be set by providing a S3 path (e.g. `s3://my-bucket/artifact_store`). A database can be created using docker, we can directly use the PostgreSQL official docker image[1] and download to the local environment. Build and run the PostgreSQL docker image, create a database for MLflow backend store and a set of credentials to access the database (e.g.database user, database password and a port) for later use. Then we use Dockerfile to install all necessary libraries and start the MLflow tracking server on the local host. The template of this Dockerfile is listed in Appendix C.2. It is an entry point to execute the `mlflow server <args>` command, the command example is also listed in Appendix C.2, it specifies what is the artifact store (i.e.the S3 path) and what is the backend store (i.e.PostgreSQL database, and all credentials). MLflow backend store supports different databases[2] and corresponding URL schema can be found as well. Now the MLflow tracking server should be accessible on localhost, by default it is `http://localhost:5000`.

Since the MLflow tracking server is up, we can run the simple Python script to try to log some parameters and artifacts to MLflow tracking. The first step, as mentioned before, is to set the MLflow URI, using Python API provided by MLflow: `mlflow.set_tracking_uri("http://localhost:5000")`[3]. Now your script knows which MLflow URI to use, the next steps is to actually log parameters. For instance, `mlflow.log_param("params_test", "success")`[4]. Finally open the localhost to check if the parameter has been logged into MLflow tracking. If successful, an experiment will show up and inside that experiment, the parameter we logged *"params_test", "success"* should exist.

4.7.3.2 Local MLflow Tracking with Azure ML

Now the MLflow tracking server is ready on localhost and the logging function is working as well. Then we can test if the local tracking server can work with Azure ML. The full example of MLflow logging function is listed in Appendix C.3. We can use those MLflow logging functions inside the ML-related scripts (e.g. `train.py`). Then submit ML jobs to Azure ML. However, there are three major problems when using separate MLflow tracking

[1] https://hub.docker.com/_/postgres
[2] https://docs.sqlalchemy.org/en/14/core/engines_connections.html
[3] https://www.mlflow.org/docs/latest/python_api/mlflow.html#mlflow.set_tracking_uri
[4] https://www.mlflow.org/docs/latest/tracking.html#id54

with Azure ML:

- The MLflow tracking URI setting inside the ML job cannot access the localhost because the ML job submitted to Azure ML runs inside the docker container and it cannot access a local server. Therefore, we use ngork[1] to expose the local server to the public internet over secure tunnels. Making our localhost as public so that it can be accessed by the docker container on Azure ML. This is also the final goal of our remote MLflow tracking server that will be introduced in the next section, Section 4.7.4. When the localhost is exposed to the public internet, use `mlflow.set_tracking_uri()` to set the tracking URI, use the URI provided by ngrok. Then the logging function is working, MLflow tracking server is working with Azure ML.

- The MLflow tracking server runs inside the Azure ML cannot access its artifact store (i.e.AWS S3). Our storage service provided by AWS is not public. We can access it in the local environment because we have AWS credentials setup in our local environment while Azure (another cloud provider) cannot. To solve this problem, we utilise Azure Key Vault[2] to keep AWS S3 access credentials (i.e.access key and secret key). Then when creating an environment for running ML jobs on Azure ML, which is via Dockerfile, mentioned in Section 4.6.2, we retrieve the AWS credentials via Azure CLI[3] and pass the credentials to Dockerfile when building the docker image. Hence, the environment where the ML job happens can have access to AWS S3. The example code of retrieving key vault from Azure is displayed in Appendix B.2.

- MLflow logging function we used inside ML-related Python scripts (i.e.`train.py`, `validate.py`), which are running on Azure ML, cannot log parameters and artifacts to local MLflow tracking. The way MLflow logging works is that MLflow will start an experiment run first when calling the Python API `mlflow.start_run()`[4], like the code example listed in Appendix C.3, then it generates a random `run_id` that uniquely identifies that run. Parameters and artifacts will be logged into corresponding experiment run on MLflow tracking UI based on that `run_id`.

  As mentioned before in Section 4.7.2, Azure ML built-in MLflow tracking will log ML entities to Azure ML workspace, directly into the correspond ML *experiment*

[1] https://ngrok.com/
[2] https://azure.microsoft.com/enau/services/keyvault/
[3] https://docs.microsoft.com/enus/azure/keyvault/secrets/quickcreatecli#retrieveasecretfromkeyvault
[4] https://www.mlflow.org/docs/latest/python_api/mlflow.html#mlflow.start_run

run (one of the components when deploying ML jobs to Azure ML, Section 4.6.2.), instead of launching a separate MLflow tracking UI. They are connected by Azure ML experiment run ID so that MLflow knows where to log the ML entities and artifacts. However, we are trying to use a separate MLflow tracking that runs outside the Azure ML. If an MLflow run starts inside Azure ML, then it creates a run that does not exist on our MLflow tracking UI. MLflow logging will complain when we are trying to use a logging function because it cannot find the corresponding run on tracking UI, therefore, it cannot log entities or artifacts to MLflow tracking. To solve this problem, we start a MLflow run locally and get the MLflow experiment `run_id`. Since the Python script we used to submit ML jobs to Azure ML (i.e.`azure_executor.py`, in Section 4.3.1) runs in local environment, we can get the `run_id` first and then pass it as the global variable to our ML scripts (e.g.`train.py`) that will be run on Azure ML. By sharing the same MLflow `run_id`, the logging that happens on Azure ML can also be logged into our MLflow tracking UI. The code example is presented in Appendix C.3.

After solving the above three problems, our local MLflow tracking server can work with Azure ML. The logging function works properly as well. It provides a good example and foundation for our remote implementation. Local implementation helps us to understand what kind of problems we might meet and what are the configuration issues between different technologies, services or even cloud providers and how to solve them.

4.7.4 Migrate MLflow Tracking Server to Cloud

The final goal of MLflow tracking is not using the local file system or docker as storage and the localhost as the URI of the tracking server. Based on the previous experience from the team, the overall design of the MLflow tracking remote server is shown in Fig 4.4. The main technologies and services are:

- **AWS Route 53**: It provides the custom domain name and DNS settings for our MLflow tracking server. We follow the similar experience of creating our custom server as we did before for our other services (e.g.Jenkins). It is the entry point for users to access our MLflow tracking server.
- **Application Load Balancer**: It helps to automatically distribute the incoming traffic across multiple targets. Load balancer acts like the traffic police, handles the requests from outside (i.e.through AWS Route 53) and coordinates the traffic on the road. It increases the availability and scalability of our tracking server.

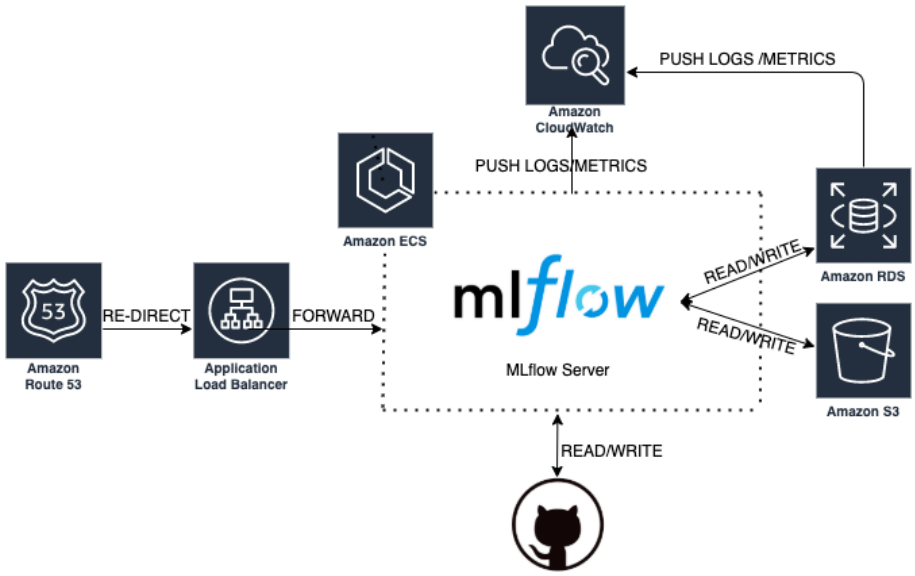

Figure 4.4: MLflow tracking server infrastructure

Then the load balancer directs the traffic to the EC2 target group[1] we specified. After that the target group routes requests to the registered target, which is our MLflow server that runs on ECS. During the process, EC2 security group is used and acts as a virtual firewall for our application. Because we do not want to open our MLflow server to the world, we add certain IPs into the security group so that they can access our MLflow server.

- **MLflow tracking server set up**: We introduced local setup for MLflow server in Section 4.7.3.1, the remote setup works similarly. We still use AWS S3 as the artifact store as it is our main storage service, but we change the database-based backend store from local docker to cloud, using AWS RDS. AWS RDS provides the easy to set up relational database service in the cloud. Price[2] for each database on RDS varies based on the settings of the database. Developers can choose different settings based on their needs. For instance, a selection of instance types is provided to fit different relational database use cases. A database can be created via AWS console, choose the database type and create a set of credentials to access the database (e.g.database user, database password and a port). The same Dockerfile is used (listed in Appendix C.2) but changing the backend setup, the database credentials, to the one we created

[1] https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-target-groups.html

[2] https://aws.amazon.com/rds/postgresql/pricing/

on AWS RDS. Then instead of running the docker image locally, we push it to AWS ECR and run it on AWS ECS.

- AWS CloudWatch: Collecting logs, metrics and events from MLflow tracking server.

With all the settings above and the local implementation experience we got, now the MLflow server is running remotely, acting as a separate service that can be accessed by the team. The MLflow logging works both locally and remotely (i.e.on Azure ML).

4.8 Integrate MLflow Tracking and DVC with ML Experiments

DVC helps us to version control the dataset and ML artefacts (e.g.model), MLflow tracking helps us to keep all the ML experiment results (e.g.metrics, parameters, artefacts). To some extent, they can both version control the ML entities(i.e. metrics) and models. The question is how to define the proper roles for them and the corresponding strategies.

In our case, we decided to use DVC as the main version control tool for ML entities and artefacts. It works similarly to Git and we follow the same Git release strategy, which we used for traditional software systems, for our ML model releases. MLflow tracking is mainly for tracking and managing each ML experiment. They work together to guarantee the traceability of our ML projects. The strategy we used for ML development workflow regarding this scenario is described in Section 4.3.2.

If the ML experiment runs in a local environment, we can directly use DVC pipelines (in Section 4.4.3) to version control the outputs (e.g.models) by adding them into DVC pipeline *outs*, then they will be tracked by DVC automatically. Finally using Git to version control relevant DVC files. DVC pipeline code examples are presented in Appendix A.2. If ML experiment happens remotely on Azure ML, those artefacts (e.g.model) are exported to Azure ML workspace after the ML algorithms run, in a special folder *./outputs* (in Section 4.6.2). We need to download them from Azure ML workspace to local project first, and then add them into DVC pipeline *outs* as well so that they can be tracked by DVC. The download code example is already listed in Appendix B.1.

5

Discussion

The ML lifecycle and MLOps architecture we proposed combined the research work we found in academic areas, technical tutorials on the marketplace and personal or the team's experience. Due to the time limit and the team's plan, there is some future work that can be done to enhance the whole infrastructure, which will be introduced in this section. We use several tools and technologies within this MLOps architecture. There are some other competitive tools that can do the same thing. The decision can be made based on the team's experience and requirements, research work about that service or tool. The discussion is also presented in this section.

5.1 Discussion

We used three different tools or technologies within our MLOps architecture. DVC (Section 4.4) is used to version control the different versions of datasets. Other data versioning tools are mentioned in Section 3.1. We choose DVC because it is an open source and it can integrate with Git easily. It does not require complicated set up and within our proposed workflow, it can handle the data versioning lifecycle for our ML projects. An important takeaway message for it is no matter what tools are used, it is important to come up with a workflow that handles the data versioning lifecycle. For instance, it should support data versioning, storing and retrieving.

For model deployment, we stick with AWS services, our main cloud provider, Sagemaker batch transform (Section 4.1.4). It is a high-performance and affordable service that is suitable for dealing with large datasets. Other approaches of deploying models into production environments are introduced in Section 3.2. MLflow can be a good candidate, since it supports the whole ML lifecycle management. We implemented model deployment

before implementing MLflow, therefore, for model deployment we stick with AWS Sagemaker. MLflow provides model registry, version control the model and specifies the stage of the model (e.g.staging, production). Then it provides API to deploy the registered model into downstream services (e.g.Sagemaker). Azure, as another cloud provider, provides all necessary services to manage the ML lifecycle. If Azure is your main cloud provider, all the ML-related services can be provided by Azure. For example, Azure ML model training we used in this graduation project (Section 4.6.2), model deployment with Azure (Section 3.2), and Azure ML also provides built-in MLflow tracking that allows you to manage all the ML experiment results (Section 4.7.2).

We choose Azure ML to help us execute the ML experiments with cloud resources. This is the team decision because of the budget but AWS also can do the same thing. One option is to use AWS Sagemaker. We use Sagemaker batch transform to help us deploy the trained model into the production environment, however, it also supports model training(19). Sagemaker model training works similarly as the Sagemaker batch transform. By providing the docker image, which contains the ML algorithm code for training the model, to Sagemaker training job, and choose the compute resources based on the requirements, Sagemaker can train the model and export the output files to AWS S3. Another option is to run the model training on ECS. We can create and build a docker image locally, which contains the ML algorithms to train the model. Then push the image to AWS ECR, and create a task definition on AWS ECS. Task definition defines which Docker image to use, how much CPU and memory to use for the task, and where to launch the task (e.g. AWS EC2). ECS makes it easy to deploy and scale Docker containers running applications, services, etc.

5.2 Reflection

This graduation project was able to successfully build a foundation for us to operationalize ML in production. As we mentioned in Section 4.1 that we emphasize on the ML development and deployment process, the data preparation and collection is out of scope. However, data plays an important role in ML projects and there are some other different use cases and challenges that are worth mentioning.

Federated learning[1], which differs from traditional large-scale machine learning. It is a technique that has multiple decentralized edge devices or servers holding local data samples. Data might spread across multiple sites. While the traditional centralized machine

[1]https://en.wikipedia.org/wiki/Federated_learning

learning techniques, all the datasets are uploaded to one server. In our case, we collect data from different online food platforms (e.g.Ubereats[1], Deliveroo[2]) and we make checkpoints for those data and keep them in a centralized place, AWS S3. However, for federated learning, the situation is different. How to manage and collect data that comes from different sites for federated machine learning use cases? Tian mentioned several challenges and methods for federated learning in their paper(20) and online blog(21). For instance, federated networks might comprise a lot of end devices (e.g.millions of smart phones), therefore **communication** in such networks can be expensive. A solution for this is to develop communication-efficient methods that iteratively send small messages as part of the model training process. Another issue is related to **data privacy**. Data generated from different sites (e.g.smartphone) might contain sensitive information. There are some methods to enhance the privacy of federated learning, using tools such as secure multi-party computation of differential privacy. However these approaches bring trade-offs, for instance, reduce model performance or system efficiency. It is important to understand these challenges and the trade-offs of bringing new approaches to solve these challenges.

In federated learning, data comes from multiple sites and can lead to a growth of data size. Even in traditional machine learning use cases, the data size can also explode. Another challenge related to this is how to automatically scale the ML development and deployment process when data size grows? Because it is hard to estimate how much resources are needed. In our case, we utilize cloud resources to help out scale the ML development and deployment process, and those cloud services have their own way of automatic scaling. For example, we use AWS Sagemaker to help us deploy the trained model into production and perform prediction logic on unseen data (Section 4.5). There could be a case that the data size grows and we are not sure how large cloud instances are needed for that ML job. However, Sagemaker supports automatic scaling for hosted models(22). Sagemaker can adjust the number of instances for the ML job based on the workload. If workload increases, this auto-scaling will bring more instances online. On the opposite site, if workload decreases, auto-scaling will reduce the number of instances to avoid wasting resources.

5.3 Future Work

We use a so-called machine learning operations maturity model provided by Azure(23) to evaluate our own MLOps framework. Based on the highlights of different levels in this

[1] https://www.ubereats.com
[2] https://deliveroo.com

maturity model and our own circumstances, we almost fulfilled the requirements of level 4. But few parts are missing and can be improved in this graduation project.

5.3.1 Automatic Remote Model Training with Jenkins

Our remote model training on Azure ML (Section 4.6) is triggered manually, by a local Python script (i.e.`azurre_executor.py`). It submits the necessary configuration of deploying custom ML models on Azure to Azure ML workspace, where the ML jobs happen and cloud resources (e.g.compute instance) can be used by ML jobs. Now, with our current setting, developers have to manually trigger it and wait in front of their laptop until the model training is done. This is not ideal for the scenario where the model training might take hours to finish. This manual triggering can be automated by adding this action into Jenkins[1] pipeline, our test automation tool used for continuous integration (CI) in software systems. We customized our own testing in Jenkins for our other software systems, such as automate building, testing library install, code unit testing, deploying, etc. Whenever a git commit is made, a pull request is opened, or a new version is released, it will automatically build the library and execute the unit tests we defined for our software systems, to ensure everything works as expected. We can automate these remote ML jobs deploying with Jenkins. Once a git commit is made, which means the ML job is ready to be deployed into Azure ML and wait for training. The whole project is packed together and submitted to Jenkins. Jenkins triggers the entry point, the Python script we used to kick off the ML pipeline, then the ML job is executed on Azure ML. During the experimentation, the same workflow happens, experiment results are logged into MLflow tracking, the ML artefacts (e.g.model) are downloaded to local once the training is done and version controlled by DVC. After that, developers can decide whether the results are satisfying enough or not. If yes, a pull request can be opened for other team members to review. If not, the same process can be executed until the satisfying results are received. By doing so, the remote ML jobs deployment is automated by Jenkins, developers do not need to manually trigger the ML pipeline and wait in front of their laptops until the ML job is done.

5.3.2 Machine Learning CI/CD Pipeline

Based on the MLOps maturity model, the CI/CD pipeline is missing in this graduation project, and it is necessary to have this CI/CD pipeline. This CI/CD pipeline forces automation in building, testing, training, retraining and even deploying. Based on the

[1] https://www.jenkins.io/

research work in Section 3.3, the report and our own MLOps framework, the CI/CD pipeline can be designed as follows.

There is an extra step before introducing CI/CD pipeline, which is a data validation test. The data validation can be done before data version control in our ML development pipeline. Data scientists are responsible for dataset creation and collecting. The data quality, for example the data distribution, can be checked by the data scientist team and then deliver the qualified dataset to the data engineer team.

Then during ML development, we work on git *feature* branch, it is a branch where you used for developing features for the ML model. For example, try out different combinations of parameters. Once a git commit is made, the code unit tests are executed, to make sure the basic function of ML algorithms is working as expected. The same tests are executed when a pull request is opened from that *feature* branch and wants to merge into the *develop* branch. Once the *feature* branch is merged into the *develop* branch, the CI pipeline triggers. It runs the same code unit tests and model training. The whole ML project is packed together and submitted to Azure ML for model training. After that, CD pipeline runs the model validation on Azure ML as well. The validation results, such as metrics, parameters will be published on MLflow tracking. Once a milestone is created, a *release* branch is published which means a new version of the model is released. Then the same CI/CD pipeline is triggered against the code based in the *release* branch. Now the model is ready to be used in production.

The similar CI/CD pipeline is used in our software systems, provided by Jenkins, but more focused on code unit testing. Whenever a git commit is made, a pull request is opened from *feature* branch, *feature* branch merges into the *develop* branch, or a *release* branch is published, the CI/CD pipeline will be triggered to validate the code quality. While in ML projects, an extra component needs to be taken care of, which is the ML model. Jenkins can also provide the automation testing. In this designed CI/CD pipeline, it supports continuous training, re-training, evaluating, and re-evaluating of ML models. At the same time, ML algorithm code is handled by this CI/CD pipeline as well.

6

Conclusion

In this thesis, we present the design of our MLOps framework and associated services or infrastructures that help us to automate and facilitate the ML lifecycle in production environments in terms of model development and model deployment. In the model development process, we utilize DVC for version control of the dataset and ML artefacts (e.g.models) that will be used or generated during ML experiments. Git is responsible for version control of the ML algorithm code and model releases. During the iterative ML development process, a separate server, MLflow Tracking, is used for managing and keeping all ML experiment results (e.g.metrics, parameters). We provide the options for users to execute their ML experiments in local environment or using cloud resources (i.e. Azure ML). They can choose based on their needs, whether the ML experiment requires more powerful compute instances or not. These services and infrastructures guaranteed the reproducibility and reliability of ML projects. Then in model deployment process, after the ML model is trained and released, we use AWS Sagemaker Batch Transform to deploy the trained model into production, its own batch strategy and the custom settings use the model we released and apply prediction logic on small or large batch dataset, which is efficient and also the price is acceptable. During this process, the released model is loaded into AWS Sagemaker during runtime, via DVC API. Since we have datasets and ML artefacts (e.g.model) under DVC control in model development process, they can be loaded into production by using DVC API, therefore, we do not need to manually handover the model and use it in production environment.

The goal of this graduation project is to establish a MLOps framework and corresponding ML development and deployment strategy that facilitate and scale ML in an industrial setting. The four research questions are revised based on the result of this thesis.

- *Q1: What are the challenges in developing machine learning projects in production?*

Challenges can be summarized into two aspects: dependency level and management level. At the dependency level, the internal dependency happens within the ML systems. ML models rely on different parts, such as data, parameters, ML algorithm code, etc. And in ML systems, unlike traditional software system, which use modular design, ML systems have no such clear boundaries. It works with data versions, algorithm code versions and different versions, different combinations of parameters will lead to different model results. The external dependency happens within the whole industrial setting. When integrating ML systems into the current industrial setting, the components around the ML system need to be taken care of. At the management level, usually in production environments, we have dozens or hundreds of models running simultaneously, the challenge is how to manage and monitor all of them? As mentioned before, ML models rely on data versions, parameter versions, and how to manage all those different versions in an efficient and reliable way is another challenge.

- *Q2: Why MLOps? What are the differences between MLOps and DevOps?*

  MLOps is a set of practices used to streamline the ML end-to-end lifecycle, designed to automate it as much as possible. MLOps and DevOps, they do share some similarities. For instance, they both encourage and facilitate collaborations between people who develop, people who manage the infrastructure and operations. They both emphasize on process automation. But for some ML-specific issues, like data versions, DevOps cannot fulfill all the requirements. That is why MLOps is introduced.

- *Q3: How to handle data versioning issues in machine learning projects and how to deploy ML models into production efficiently?*

  Data versioning is a big challenge in ML projects. There are different tools on the marketplace to help us solve this problem. As mentioned in Section 3.1, and the one we used, DVC, they can help us version the data overtime and keep them in a reliable place (e.g.remote storage). The challenge is to find the proper tool, integrate with current industrial settings and create relevant strategies for data versioning.

  Model is deployed into production environments utilizing cloud resources. Different cloud providers have their own way of deploying ML models. In our case, we use AWS Sagemaker Batch Transform. The decision can be made based on the team's previous experience and requirements. Using cloud resources are more reliable and scalable since cloud can provides the compute power that your local environment might not

equipped. Within a team, it also defines the standard way of deploying models into production, otherwise, team members might have their own ways of deploying which lead to inconsistency.

- *Q4: What kind of MLOps frameworks or strategies that we could learn from when integrating ML systems into the industrial setting?*

  The MLOps framework is aiming to automate and facilitate the entire machine learning lifecycle, which enables continuous development and continuous delivery. We emphasize on the model development and model deployment process. We present a framework and strategies that allow us to continuously develop ML models and release them into production. Different versions of data are properly handled and saved. The iterative ML development process generates a lot of experiment results, which are also carefully tracked and managed. This enables us to continuous develop and release in a reliable and efficient way. Our MLOps framework also builds a bridge between model development and model deployment. Released models can be deployed into production environments regarding the model version, without any manual handover steps.

The purpose of this graduation project and thesis is to provide a template framework and workflow strategy for those people or teams who are interested in MLOps and trying to build their own MLOps framework. We started with no MLOps at all, now we have a standard and basic MLOps architecture and workflow strategy that help us manage our ML development and deployment process, also the similar architecture and workflow are applied to our other ML projects. This MLOps architecture fulfilled our key objectives. A standard ML lifecycle has been established. Our data scientist team and data engineer team can have a better collaboration on ML projects. Data, ML artefacts lifecycle, and the ML experiment details have been carefully tracked and saved. No manual handover is needed when deploying models into production. It streamlines and automates the ML lifecycle, reduces the delivery time and labour work, makes the ML development process more reliable, traceable and scalable.

Appendix

A DVC Code Examples

A.1 DVC Build Pipeline Example

This code examples are for Section 4.3.2. The full process (i.e. train and validate) stage in DVC pipeline can be created by following code. It specifies the dependencies (`-d`), outputs (`-o`), metrics (`-M`) and plots (`-plots-no-cache`). Run the command from the directory dvc_pipeline/NL:

```
dvc run train_validate_local \
-d ../../src/dash_ml_flag_combo/scripts/run_pipeline.py \
-d ../../assets/data/NL/NL_training.csv \
-d ../../assets/data/NL/NL_validation.csv \
-o ../../assets/model/NL/NL_flag_combo.pkl \
-M ../../assets/metrics/NL/NL_scoring.json \
-plots-no-cache ../../assets/metrics/NL/NL_confusion_matrix.png \
python ../../src/dash_ml_flag_combo/scripts/run_pipeline.py ${params.country} \
${mode.train_validate} ${pipeline.local}
```

The validate stage can be created by the following code, following the same rules:

```
dvc run validate_local \
-d ../../src/dash_ml_flag_combo/scripts/run_pipeline.py \
-d ../../assets/data/NL/NL_validation.csv \
-o ../../assets/data/NL/NL_validated.csv \
python ../../src/dash_ml_flag_combo/scripts/run_pipeline.py ${params.country} \
${mode.validate} ${pipeline.local}
```

A.2 DVC Pipeline File Examples

The generated DVC pipeline related files, creating by the code above in Section A.1, are presented here, along with the *params.yaml* file.

- ***dvc.yaml***[1] **file**:

```
stages:
  train_validate_local:
    cmd: python ../../src/dash_ml_flag_combo/scripts/run_pipeline.py ${params.country} \
    ${mode.train_validate} ${pipeline.local}
    deps:
    - ../../assets/data/NL/NL_training.csv
    - ../../assets/data/NL/NL_validation.csv
    - ../../src/dash_ml_flag_combo/scripts/run_pipeline.py
    outs:
    - ../../assets/model/NL/NL_flag_combo.pkl
    metrics:
      - ../../assets/metrics/NL/NL_scoring.json:
          cache: false
    plots:
      - ../../assets/metrics/NL/NL_confusion_matrix.png:
          cache: false
  validate_local:
    cmd: python ../../src/dash_ml_flag_combo/scripts/run_pipeline.py ${params.country} \
    ${mode.validate} ${pipeline.local}
    deps:
    - ../../assets/data/NL/NL_validation.csv
    - ../../src/dash_ml_flag_combo/scripts/run_pipeline.py
    outs:
    - ../../assets/data/NL/NL_validated.csv
```

- ***dvc.lock***[2] **file**:

```
schema: '2.0'
stages:
  train_validate_local:
```

[1] https://dvc.org/doc/user-guide/project-structure/pipelines-files#pipelines-files-dvcyaml
[2] https://dvc.org/doc/user-guide/how-to/merge-conflicts#dvclock

```
    cmd: python ../../src/dash_ml_flag_combo/scripts/run_pipeline.py NL train_validate \
    local_run
    deps:
    - path: ../../assets/data/NL/NL_training.csv
      md5: 60b27ce834dad76825fn96b26edacc9b
      size: 68595
    - path: ../../assets/data/NL/NL_validation.csv
      md5: 0c80660e4eaaea3b7337y13de2097506
      size: 35666
    - path: ../../src/dash_ml_flag_combo/scripts/run_pipeline.py
      md5: ef1658ob652138pd0125d14983de6809
      size: 602
    outs:
    - path: ../../assets/metrics/NL/NL_confusion_matrix.png
      md5: d60480r7a5c2f9033k534660b5d631eb
      size: 14634
    - path: ../../assets/metrics/NL/NL_scoring.json
      md5: 5d1ad161l06df2ecdd778d8fab79f760
      size: 128
    - path: ../../assets/model/NL/NL_flag_combo.pkl
      md5: 48b8516yh274927617a985v8540915ea
      size: 693806
  validate_local:
    cmd: python ../../src/dash_ml_flag_combo/scripts/run_pipeline.py NL validate \
    local_run
    deps:
    - path: ../../assets/data/NL/NL_validation.csv
      md5: 0c84360e4loefa3b7398e13de2097506
      size: 35666
    - path: ../../src/dash_ml_flag_combo/scripts/run_pipeline.py
      md5: ef1047ac052138fs0425d14983de6809
      size: 602
    outs:
    - path: ../../assets/data/NL/NL_validated.csv
      md5: 7ed1d0g18b66b6e4d9577c7b20c6a2dd1
      size: 20694
```

- ***params.yaml***[1] **file**:

```
params:
  country: "NL"
mode:
  train_validate: "train_validate"
  validate: "validate"
pipeline:
  local: "local_run"
  remote: "azure_run"
```

B Azure ML Related Code Examples

B.1 ML Jobs with Azure ML Configuration

- **Create Azure ML Workspace**:

```
from azureml.core import Workspace
ws = Workspace.create(name='myworkspace',
          subscription_id='<azure-subscription-id>',
          resource_group='myresourcegroup',
          create_resource_group=True,
          location='eastus2'
         )
```

- **Create compute instance**:

```
from azureml.core.compute import ComputeTarget, ComputeInstance
from azureml.core.compute_target import ComputeTargetException
compute_name = "mycomputeinstance"
# Check if the instance already exist
try:
    instance = ComputeInstance(workspace=ws, name=compute_name)
    print('Found existing instance, use it.')
except ComputeTargetException:
    compute_config = ComputeInstance.provisioning_configuration(
        vm_size='STANDARD_D3_V2',
        ssh_public_access=False
    )
```

[1] https://dvc.org/doc/command-reference/params#params

```
compute_instance = ComputeInstance.create(ws, compute_name, compute_config)
compute_instance.wait_for_completion(show_output=True)
```

- **Create experiment**:

```
from azureml.core import Experiment
experiment_name = 'myexperiment'
experiment = Experiment(workspace=ws, name=experiment_name)
```

- **Using custom docker image from ACR as environment**:

```
from azureml.core import Environment
# load existing image from ACR
my_env = Environment(name='docker_image_environment')
my_env.docker.base_image = IMAGE_PATH_ON_ACR
my_env.python.user_managed_dependencies = True
```

- **Script run configuration and submit job**:

```
from azureml.core import ScriptRunConfig
src = ScriptRunConfig(
            source_directory=project_folder,
            script=python_script,
            compute_target=compute_instance,
            environment=my_env,
            arguments=[arg_1, arg_2, arg_n])
run = experiment.submit(config=src)
run.wait_for_completion(show_output=True)
```

- **Download ML artefacts from *./outputs* folder to local**:

```
model_output_path = "PATH_IN_LOCAL"
run.download_file(
        name="./outputs/model.pkl",
        output_file_path=model_output_path,
        )
```

B.2 Retrieving Key Vault

```
az keyvault secret show --name "aws-s3-access-key" \
--vault-name "<your-unique-keyvault-name>" --query "value"
```

C MLflow Tracking Code Examples

C.1 MLflow User Interface

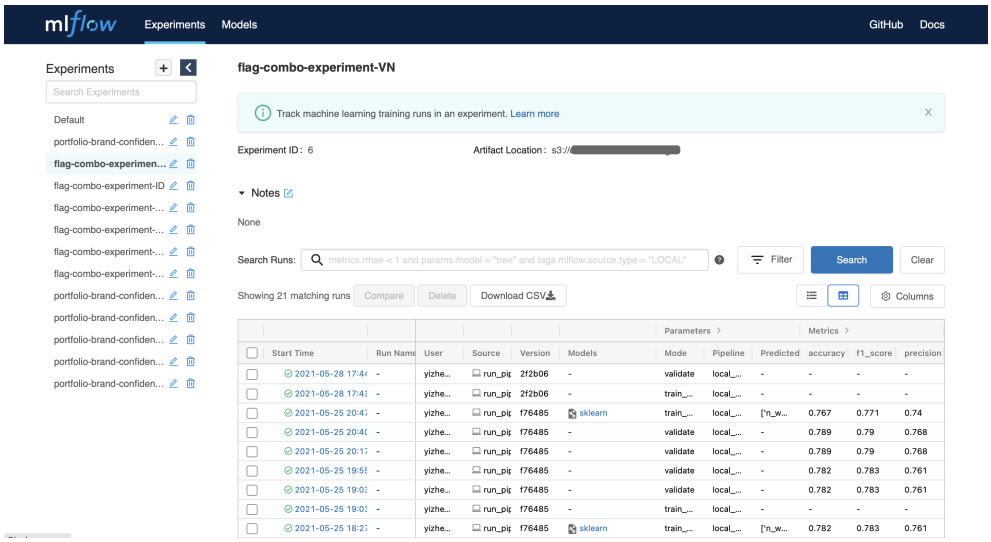

Figure 6.1: MLflow Tracking Interface (with different experiments)

Figure 6.2: One experiment details in MLflow Tracking

C.2 MLflow Tracking Server Local Setup

- **Dockerfile**:

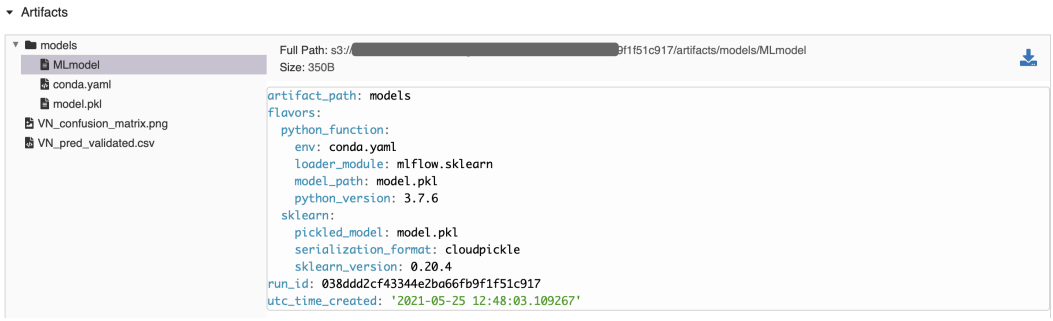

Figure 6.3: MLflow Model, model has been logged into MLflow Tracking. It describes details of the ML model, prerequisite of Model Registry

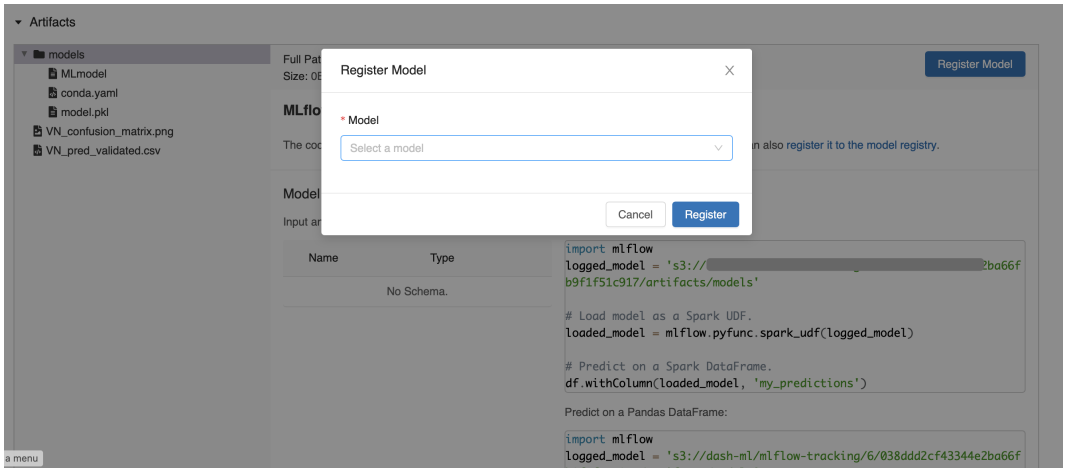

Figure 6.4: Model Registry, registered model will be kept in a centralized place. Shown in next Figure

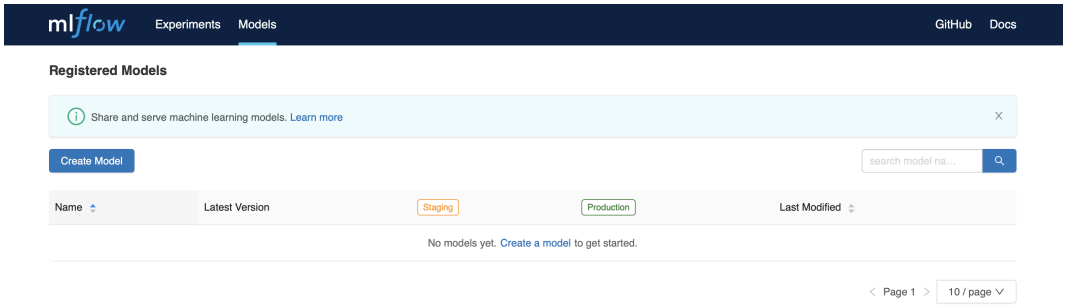

Figure 6.5: Centralized place for registered models, with stages of each model (e.g.staging, production).

```
FROM python:3.7
RUN pip install mlflow==1.13 &&
    pip install awscli --upgrade --user &&
    pip install boto3==1.16.46 &&
    pip install psycopg2-binary

# defining workdir
WORKDIR /home/ubuntu
COPY files/run.sh /home/ubuntu
RUN chmod +x /home/ubuntu/run.sh
ENTRYPOINT ["/home/ubuntu/run.sh"]
```

- **MLflow configration command (inside the file *run.sh*)**:

```
mlflow server \
    --backend-store-uri postgresql://DB_USER:DB_PASSWORD@localhost/DB_name \
    --default-artifact-root s3://my-bucket/artifact_store \
    --host 0.0.0.0
```

C.3 MLflow Logging

- **MLflow logging functions**:

```
mlflow.set_tracking_uri("http://localhost:5000")
# Start a MLflow run/experiment
with mlflow.start_run():
    mlflow.log_param("data_url", data_url)
    mlflow.log_metrics(metrics)
    mlflow.log_artifact(data_path)
```

- **MLflow logging sharing same run_id**:

```
mlflow.set_tracking_uri("<mlflow-server-uri>")
# Start a MLflow run/experiment in a local script (e.g.azure_executor.py)
with mlflow.start_run():
    # Get active run_id
    run = mlflow.active_run()
    mlflow_run_id = run.info.run_id
    src = ScriptRunConfig(
        source_directory=project_folder,
```

```
            script="train.py",
            compute_target=compute_instance,
            environment=my_env,
            arguments=[mlflow_run_id]) # Pass run_id to train.py as global variable
        run = experiment.submit(config=src)
        run.wait_for_completion(show_output=True)

    # Using MLflow logging in train.py, which runs on Azure ML
    with mlflow.start_run(run_id=mlflow_run_id):
        mlflow.log_param("data_url", data_url)
        mlflow.log_metrics(metrics)
        mlflow.log_artifact(data_path)
```

References

[1] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. **Hidden Technical Debt in Machine Learning Systems**. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, page 2503–2511, Cambridge, MA, USA, 2015. MIT Press. 1

[2] Yizhen Zhao. **Machine Learning in Production: A Literature Review**. 3

[3] Andrei Paleyes, Raoul-Gabriel Urma, and Neil D. Lawrence. **Challenges in Deploying Machine Learning: a Survey of Case Studies**. *arXiv e-prints*, page arXiv:2011.09926, November 2020. 5

[4] Adarsh Shah. **Challenges Deploying Machine Learning Models to Production**. 5

[5] Luigi. **5 Challenges to Running Machine Learning Systems in Production**. 5

[6] Anant Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J. Elmore, Samuel Madden, and Aditya G. Parameswaran. **DataHub: Collaborative Data Science Dataset Version Management at Scale**, 2014. 7

[7] Vimarsh Karbhari. **MLOps: Data Science Version Control**. 8

[8] Azure. **Deploy machine learning models to Azure**. 9

[9] Kyle Gallatin. **Deploying Models to Production with Mlflow and Amazon Sagemaker**. 10

[10] Azure. **Deploy MLflow models as Azure web services**. 10

[11] Databricks. **MLflow Quick Start Part 2: Serving Models with Microsoft Azure ML**. 10

[12] Emmanuel Raj. **Edge MLOps framework for AIoT applications, Continuous delivery for AIoT, Big Data and 5G applications**. June 2020. 10

[13] István Pölöskei. **MLOps approach in the cloud-native data pipeline design**. *Acta Technica Jaurinensis*, 04 2021. 11

[14] Yizhen Zhao. **MLOps and data versioning in machine learning project**. 21, 24

[15] Yizhen Zhao. **MLOps: Data versioning with DVC — Part I**. 21

[16] Yizhen Zhao. **MLOps: Deploy custom model with AWS Sagemaker batch transform — Part II**. 24

[17] MLflow. **How Runs and Artifacts are Recorded**. 29

[18] Azure. **Track ML models with MLflow and Azure Machine Learning**. 29

[19] AWS. **Train a Model with Amazon SageMaker**. 37

[20] Tian Li, Anit Kumar Sahu, Ameet S. Talwalkar, and Virginia Smith. **Federated Learning: Challenges, Methods, and Future Directions**. *IEEE Signal Processing Magazine*, **37**:50–60, 2020. 38

[21] Tian Li. **Federated Learning: Challenges, Methods, and Future Directions**. 38

[22] AWS. **Automatically Scale Amazon SageMaker Models**. 38

[23] Azure. **Machine Learning Operations maturity model**. 38